**Superset ID- 6363144**

**Name- Harshita Sinha**

**Week 1- Engineering Concepts (Algorithms\_Data-Structures)**

**Exercise 1: Inventory Management System**

**SOLUTION:-**

using System;

using System.Collections.Generic;

namespace InventoryManagement

{

    public class Product

    {

        public int ProductId { get; set; }

        public string ProductName { get; set; }

        public int Quantity { get; set; }

        public double Price { get; set; }

        public Product(int id, string name, int quantity, double price)

        {

            ProductId = id;

            ProductName = name;

            Quantity = quantity;

            Price = price;

        }

        public override string ToString()

        {

            return $"ID: {ProductId}, Name: {ProductName}, Quantity: {Quantity}, Price: ₹{Price}";

        }

    }

    public class Inventory

    {

        private Dictionary<int, Product> products = new Dictionary<int, Product>();

        public void AddProduct(Product product)

        {

            if (products.ContainsKey(product.ProductId))

            {

                Console.WriteLine("Product ID already exists. Use update instead.");

                return;

            }

            products[product.ProductId] = product;

            Console.WriteLine("Product added successfully.");

        }

        public void UpdateProduct(int id, int newQuantity, double newPrice)

        {

            if (products.ContainsKey(id))

            {

                products[id].Quantity = newQuantity;

                products[id].Price = newPrice;

                Console.WriteLine("Product updated successfully.");

            }

            else

            {

                Console.WriteLine("Product not found.");

            }

        }

        public void DeleteProduct(int id)

        {

            if (products.Remove(id))

            {

                Console.WriteLine("Product deleted successfully.");

            }

            else

            {

                Console.WriteLine("Product not found.");

            }

        }

        public void DisplayInventory()

        {

            Console.WriteLine("\nCurrent Inventory:");

            foreach (var product in products.Values)

            {

                Console.WriteLine(product);

            }

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            Inventory inventory = new Inventory();

            inventory.AddProduct(new Product(101, "Mouse", 25, 499.99));

            inventory.AddProduct(new Product(102, "Keyboard", 15, 999.99));

            inventory.UpdateProduct(101, 30, 450.00);

            inventory.DeleteProduct(102);

            inventory.DisplayInventory();

            inventory.AddProduct(new Product(107, "Digital Pen", 19, 399.99));

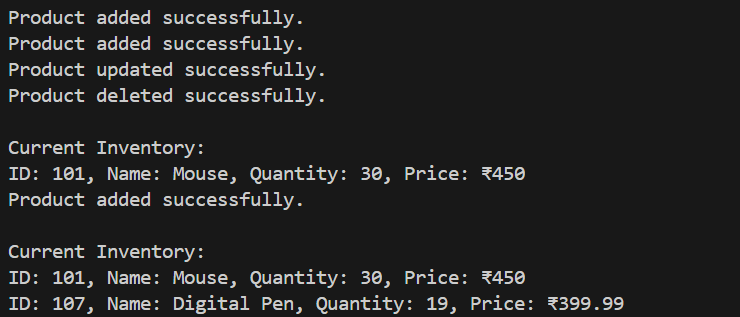
            inventory.DisplayInventory();

        }

    }

}

**OUTPUT:-**



**ANALYSIS:-**

* Time Complexities: AddProduct, UpdateProduct, DeleteProduct: O(1) (average case due to hash map)
* DisplayInventory: O(n) (where n is the number of products)
* Optimization Notes:

Already optimal with a dictionary.

For large systems: consider adding persistence (database) and concurrency control.

**Exercise 2: E-commerce Platform Search Function**

**SOLUTION:-**

using System;

using System.Linq;

public class Product

{

public int ProductId { get; set; }

public string ProductName { get; set; }

public string Category { get; set; }

public Product(int id, string name, string category)

{

ProductId = id;

ProductName = name;

Category = category;

}

public override string ToString()

{

return $"ID: {ProductId}, Name: {ProductName}, Category: {Category}";

}

}

class Program

{

static void Main(string[] args)

{

Product[] products = new Product[]

{

new Product(1, "Laptop", "Electronics"),

new Product(2, "Shirt", "Clothing"),

new Product(3, "Phone", "Electronics"),

new Product(4, "Shoes", "Footwear")

};

Console.WriteLine("\n🔍 Linear Search:");

var foundLinear = LinearSearch(products, "Phone");

Console.WriteLine(foundLinear != null ? $"Found: {foundLinear}" : "Not Found");

Console.WriteLine("\n🔍 Binary Search (sorted array):");

var sortedProducts = products.OrderBy(p => p.ProductName).ToArray();

var foundBinary = BinarySearch(sortedProducts, "Phone");

Console.WriteLine(foundBinary != null ? $"Found: {foundBinary}" : "Not Found");

}

public static Product LinearSearch(Product[] products, string searchName)

{

foreach (Product p in products)

{

if (p.ProductName.Equals(searchName, StringComparison.OrdinalIgnoreCase))

return p;

}

return null;

}

public static Product BinarySearch(Product[] products, string searchName)

{

int left = 0;

int right = products.Length - 1;

while (left <= right)

{

int mid = (left + right) / 2;

int compare = string.Compare(products[mid].ProductName, searchName, StringComparison.OrdinalIgnoreCase);

if (compare == 0)

return products[mid];

else if (compare < 0)

left = mid + 1;

else

right = mid - 1;

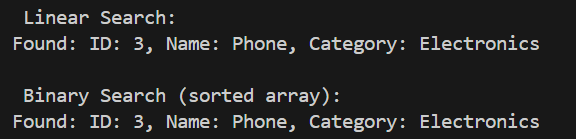
}

return null;

}

}

**OUTPUT**



**ANALYSIS:-**

* Linear Search: O(n)
* Binary Search: O(log n) (but requires sorted data)
* Optimization Notes: Binary search is faster for large sorted datasets.

**Exercise 3: Sorting Customer Orders**

**SOLUTION:-**

using System;

public class Order

{

public int OrderId { get; set; }

public string CustomerName { get; set; }

public double TotalPrice { get; set; }

public Order(int id, string name, double price)

{

OrderId = id;

CustomerName = name;

TotalPrice = price;

}

public override string ToString()

{

return $"OrderID: {OrderId}, Customer: {CustomerName}, Total: ₹{TotalPrice}";

}

}

class Program

{

static void Main()

{

Order[] orders = new Order[]

{

new Order(1, "Harshita", 3200.50),

new Order(2, "Soumya", 1450.00),

new Order(3, "Rupsa", 5890.99),

new Order(4, "Shruti", 2200.00),

new Order(5, "Rishu", 5890.99)

};

Console.WriteLine("\n Original Orders:");

PrintOrders(orders);

Order[] bubbleSorted = (Order[])orders.Clone();

BubbleSort(bubbleSorted);

Console.WriteLine("\n Orders Sorted by Bubble Sort (Ascending by TotalPrice):");

PrintOrders(bubbleSorted);

Order[] quickSorted = (Order[])orders.Clone();

QuickSort(quickSorted, 0, quickSorted.Length - 1);

Console.WriteLine("\n Orders Sorted by Quick Sort (Ascending by TotalPrice):");

PrintOrders(quickSorted);

}

public static void PrintOrders(Order[] orders)

{

foreach (var order in orders)

{

Console.WriteLine(order);

}

}

public static void BubbleSort(Order[] orders)

{

int n = orders.Length;

for (int i = 0; i < n - 1; i++)

{

bool swapped = false;

for (int j = 0; j < n - i - 1; j++)

{

if (orders[j].TotalPrice > orders[j + 1].TotalPrice)

{

var temp = orders[j];

orders[j] = orders[j + 1];

orders[j + 1] = temp;

swapped = true;

}

}

if (!swapped) break;

}

}

public static void QuickSort(Order[] orders, int low, int high)

{

if (low < high)

{

int pivotIndex = Partition(orders, low, high);

QuickSort(orders, low, pivotIndex - 1);

QuickSort(orders, pivotIndex + 1, high);

}

}

private static int Partition(Order[] orders, int low, int high)

{

double pivot = orders[high].TotalPrice;

int i = low - 1;

for (int j = low; j < high; j++)

{

if (orders[j].TotalPrice < pivot)

{

i++;

var temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

var temp1 = orders[i + 1];

orders[i + 1] = orders[high];

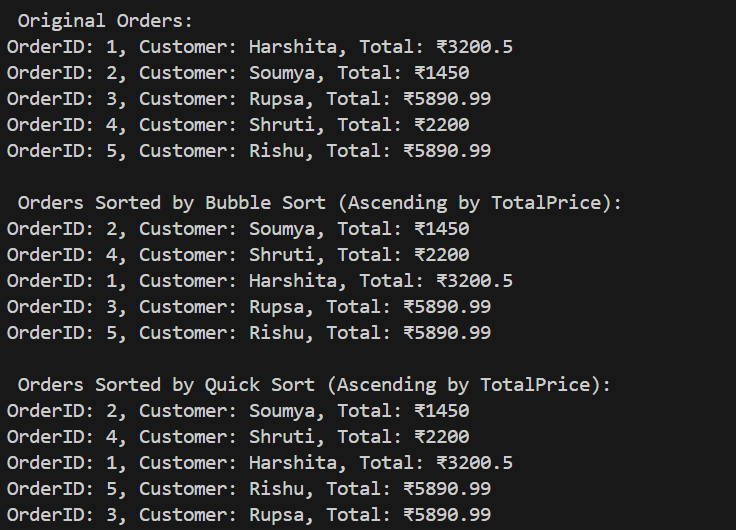
orders[high] = temp1;

return i + 1;

}

}

**OUTPUT:-**



**ANALYSIS:-**

* Bubble Sort: O(n²) – inefficient for large lists.
* Quick Sort: Average case O(n log n), worst case O(n²)
* Optimization Notes: QuickSort is significantly faster in practice.

For production, use Array.Sort() or LINQ’s .OrderBy() (internally optimized).

**Exercise 4: Employee Management System**

**SOLUTION:-**

using System;

public class Employee

{

public int EmployeeId { get; set; }

public string Name { get; set; }

public string Position { get; set; }

public double Salary { get; set; }

public Employee(int id, string name, string position, double salary)

{

EmployeeId = id;

Name = name;

Position = position;

Salary = salary;

}

public override string ToString()

{

return $"ID: {EmployeeId}, Name: {Name}, Position: {Position}, Salary: ₹{Salary}";

}

}

public class EmployeeManager

{

private Employee[] employees;

private int count;

public EmployeeManager(int size)

{

employees = new Employee[size];

count = 0;

}

public void AddEmployee(Employee emp)

{

if (count >= employees.Length)

{

Console.WriteLine("Employee list is full. Cannot add more.");

return;

}

employees[count++] = emp;

Console.WriteLine(" Employee added successfully.");

}

public Employee SearchEmployee(int id)

{

for (int i = 0; i < count; i++)

{

if (employees[i].EmployeeId == id)

return employees[i];

}

return null;

}

public void TraverseEmployees()

{

if (count == 0)

{

Console.WriteLine("No employee records found.");

return;

}

Console.WriteLine("\n Employee Records:");

for (int i = 0; i < count; i++)

{

Console.WriteLine(employees[i]);

}

}

public void DeleteEmployee(int id)

{

int index = -1;

for (int i = 0; i < count; i++)

{

if (employees[i].EmployeeId == id)

{

index = i;

break;

}

}

if (index == -1)

{

Console.WriteLine(" Employee not found.");

return;

}

for (int i = index; i < count - 1; i++)

{

employees[i] = employees[i + 1];

}

employees[count - 1] = null;

count--;

Console.WriteLine("🗑️ Employee deleted successfully.");

}

}

class Program

{

static void Main()

{

EmployeeManager manager = new EmployeeManager(5);

manager.AddEmployee(new Employee(101, "Soumya", "Manager", 75000));

manager.AddEmployee(new Employee(102, "Harshita", "Developer", 60000));

manager.AddEmployee(new Employee(103, "Rupsa", "Designer", 55000));

manager.TraverseEmployees();

Console.WriteLine("\n Searching for Employee ID 102:");

var emp = manager.SearchEmployee(102);

Console.WriteLine(emp != null ? $"Found: {emp}" : "Not found");

Console.WriteLine("\n Deleting Employee ID 101:");

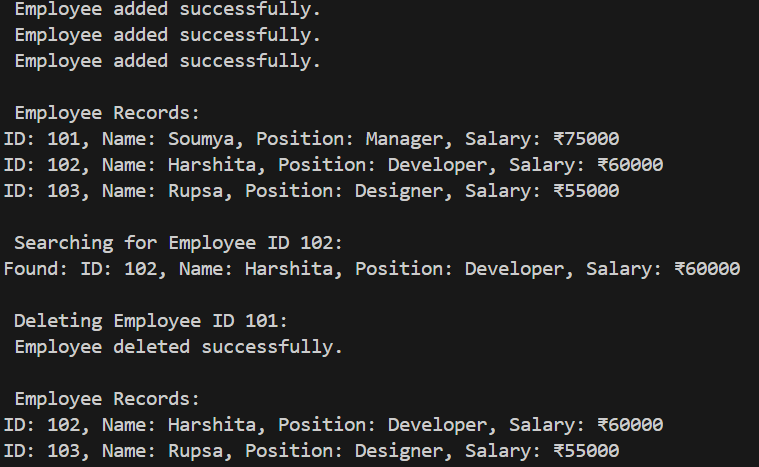
manager.DeleteEmployee(101);

manager.TraverseEmployees();

}

}

**OUTPUT:-**



**ANALYSIS:-**

* Data Structure Used: Array
* Time Complexities: Add: O(1),Search: O(n), Traverse: O(n), Delete: O(n) (due to shifting)
* Optimization Notes:

Arrays are fixed-size and inefficient for deletions.

Use List<Employee> for dynamic resizing and simplified management.

**Exercise 5: Task Management System**

**SOLUTION:-**

using System;

public class Task

{

public int TaskId { get; set; }

public string TaskName { get; set; }

public string Status { get; set; }

public Task(int id, string name, string status)

{

TaskId = id;

TaskName = name;

Status = status;

}

public override string ToString()

{

return $"TaskID: {TaskId}, Name: {TaskName}, Status: {Status}";

}

}

public class TaskNode

{

public Task Data { get; set; }

public TaskNode Next { get; set; }

public TaskNode(Task task)

{

Data = task;

Next = null;

}

}

public class TaskManager

{

private TaskNode head;

public TaskManager()

{

head = null;

}

public void AddTask(Task task)

{

TaskNode newNode = new TaskNode(task);

if (head == null)

{

head = newNode;

}

else

{

TaskNode current = head;

while (current.Next != null)

{

current = current.Next;

}

current.Next = newNode;

}

Console.WriteLine(" Task added successfully.");

}

public void TraverseTasks()

{

if (head == null)

{

Console.WriteLine("No tasks found.");

return;

}

Console.WriteLine("\n Task List:");

TaskNode current = head;

while (current != null)

{

Console.WriteLine(current.Data);

current = current.Next;

}

}

public Task SearchTask(int id)

{

TaskNode current = head;

while (current != null)

{

if (current.Data.TaskId == id)

return current.Data;

current = current.Next;

}

return null;

}

public void DeleteTask(int id)

{

if (head == null)

{

Console.WriteLine("No tasks to delete.");

return;

}

if (head.Data.TaskId == id)

{

head = head.Next;

Console.WriteLine(" Task deleted successfully.");

return;

}

TaskNode current = head;

TaskNode prev = null;

while (current != null && current.Data.TaskId != id)

{

prev = current;

current = current.Next;

}

if (current == null)

{

Console.WriteLine(" Task not found.");

return;

}

prev.Next = current.Next;

Console.WriteLine(" Task deleted successfully.");

}

}

class Program

{

static void Main()

{

TaskManager manager = new TaskManager();

manager.AddTask(new Task(1, "Design Database", "Pending"));

manager.AddTask(new Task(2, "Develop API", "In Progress"));

manager.AddTask(new Task(3, "Write Documentation", "Pending"));

manager.TraverseTasks();

Console.WriteLine("\n Searching for Task ID 2:");

var found = manager.SearchTask(2);

Console.WriteLine(found != null ? $"Found: {found}" : "Not Found");

Console.WriteLine("\n Deleting Task ID 1:");

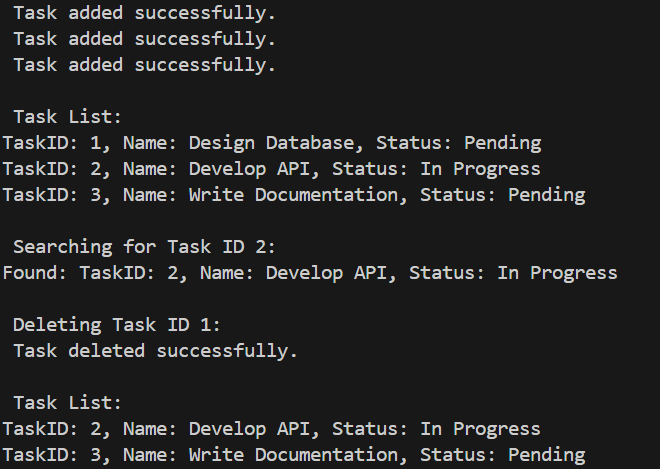
manager.DeleteTask(1);

manager.TraverseTasks();

}

}

**OUTPUT:-**



**ANALYSIS:-**

* Time Complexities:Add: O(n) (at end), Search: O(n), Traverse: O(n), Delete: O(n)
* Optimization Notes:

Use LinkedList<T> from .NET if bi-directional traversal or head/tail insertions are needed.

Better for frequent insert/delete than arrays

**Exercise 6: Library Management System**

**SOLUTION:-**

using System;

using System.Collections.Generic;

public class Book

{

public int BookId { get; set; }

public string Title { get; set; }

public string Author { get; set; }

public Book(int id, string title, string author)

{

BookId = id;

Title = title;

Author = author;

}

public override string ToString()

{

return $"ID: {BookId}, Title: {Title}, Author: {Author}";

}

}

public class Library

{

private List<Book> books = new List<Book>();

public void AddBook(Book book)

{

books.Add(book);

}

public void ShowBooks()

{

Console.WriteLine("\n Book List:");

foreach (var book in books)

{

Console.WriteLine(book);

}

}

public Book LinearSearchByTitle(string title)

{

foreach (var book in books)

{

if (book.Title.Equals(title, StringComparison.OrdinalIgnoreCase))

{

return book;

}

}

return null;

}

public Book BinarySearchByTitle(string title)

{

books.Sort((b1, b2) => b1.Title.CompareTo(b2.Title));

int low = 0;

int high = books.Count - 1;

while (low <= high)

{

int mid = (low + high) / 2;

int cmp = string.Compare(title, books[mid].Title, StringComparison.OrdinalIgnoreCase);

if (cmp == 0)

{

return books[mid];

}

else if (cmp < 0)

{

high = mid - 1;

}

else

{

low = mid + 1;

}

}

return null;

}

}

class Program

{

static void Main()

{

Library library = new Library()

library.AddBook(new Book(101, "C# Programming", "John Smith"));

library.AddBook(new Book(102, "Data Structures", "Jane Doe"));

library.AddBook(new Book(103, "Operating Systems", "Andrew Tanenbaum"));

library.AddBook(new Book(104, "Algorithms Unlocked", "Thomas Cormen"));

library.AddBook(new Book(105, "Design Patterns", "Erich Gamma"));

library.ShowBooks();

Console.WriteLine("\n Linear Search: 'Data Structures'");

var result1 = library.LinearSearchByTitle("Data Structures");

Console.WriteLine(result1 != null ? $" Found: {result1}" : " Book not found");

Console.WriteLine("\n Binary Search: 'Operating Systems'");

var result2 = library.BinarySearchByTitle("Operating Systems");

Console.WriteLine(result2 != null ? $"Found: {result2}" : "Book not found");

Console.WriteLine("\n Binary Search: 'Unknown Book'");

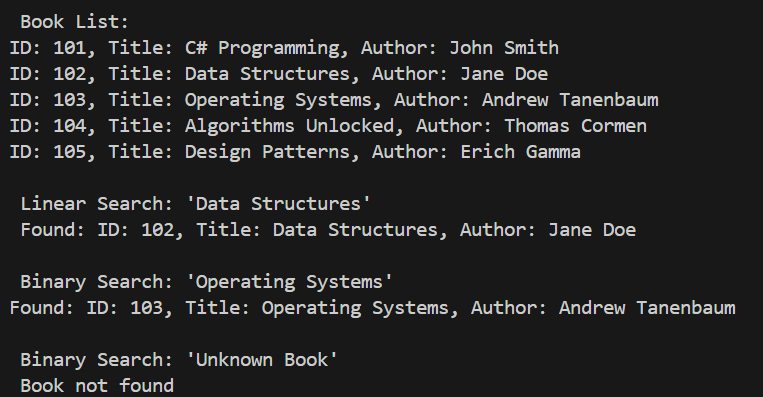
var result3 = library.BinarySearchByTitle("Unknown Book");

Console.WriteLine(result3 != null ? $" Found: {result3}" : " Book not found");

}

}

**OUTPUT:-**



**ANALYSIS:-**

* Linear Search: O(n)
* Binary Search: O(log n) (on sorted list)
* Optimization Notes:

For static data or frequent search: use binary search after sorting.

For dynamic or large data: use dictionary or indexing structure.

**Exercise 7: Financial Forecasting**

**SOLUTION:-**

using System;

public class FinancialForecaster

{

public static double PredictFutureValue(double initialValue, double growthRate, int years)

{

if (years == 0)

{

return initialValue;

}

return PredictFutureValue(initialValue, growthRate, years - 1) \* (1 + growthRate);

}

public static double PredictFutureValueMemo(double initialValue, double growthRate, int years, double[] memo)

{

if (years == 0)

{

return initialValue;

}

if (memo[years] != 0)

{

return memo[years];

}

memo[years] = PredictFutureValueMemo(initialValue, growthRate, years - 1, memo) \* (1 + growthRate);

return memo[years];

}

}

class Program

{

static void Main()

{

double initialInvestment = 10000;

double annualGrowthRate = 0.08;

int targetYear = 5;

Console.WriteLine($" Forecasting financial value for {targetYear} years...");

double result = FinancialForecaster.PredictFutureValue(initialInvestment, annualGrowthRate, targetYear);

Console.WriteLine($"\n Recursive Result (Year {targetYear}): ₹{result:F2}");

double[] memo = new double[targetYear + 1];

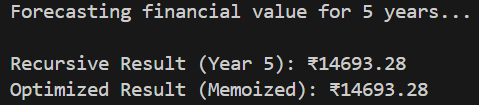
double resultMemo = FinancialForecaster.PredictFutureValueMemo(initialInvestment, annualGrowthRate, targetYear, memo);

Console.WriteLine($" Optimized Result (Memoized): ₹{resultMemo:F2}");

}

}

**OUTPUT:-**



**ANALYSIS:-**

Recursive Version: O(n) time, O(n) space (due to call stack)

Memoized Version: O(n) time, O(n) space

Optimization Notes:

Memoization prevents repeated calculations.

Can also implement iteratively for O(n) time and O(1) space.